Creating a language using only assembly language.

Kernel/VM Tanken-tai #11
Koichi Nakamura

Codes

- https://github.com/nineties/amber

Profile

- Koichi Nakamura
  - twitter: @9_ties
- developing an IoT device
  - http://idein.jp

I was a compiler writer

- wrote compilers at student experiment
  - minCaml compiler by O'Caml
  - minCaml compiler by Haskell
    https://github.com/nineties/Choco

- studied optimizing compilers at graduate school
  - wrote compilers for special purpose CPUs

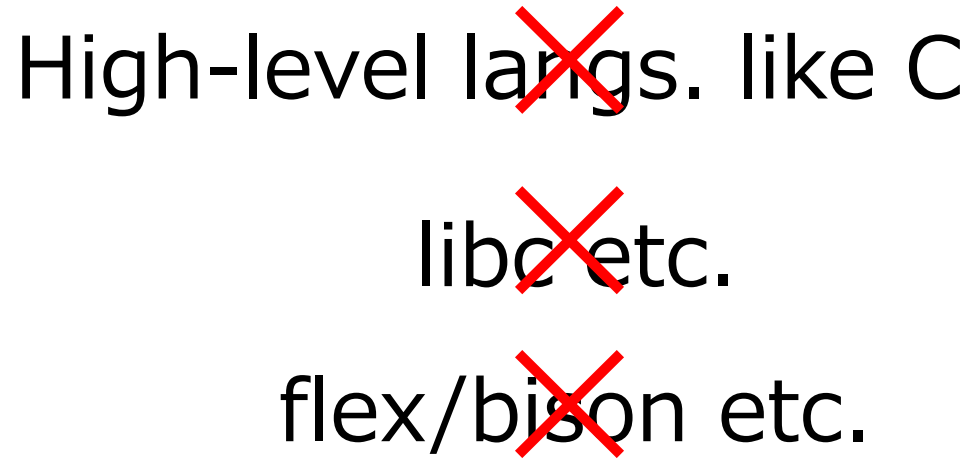
Wanted to create my own language

- name: “Amber”
  - It was ‘“rowl” at first.

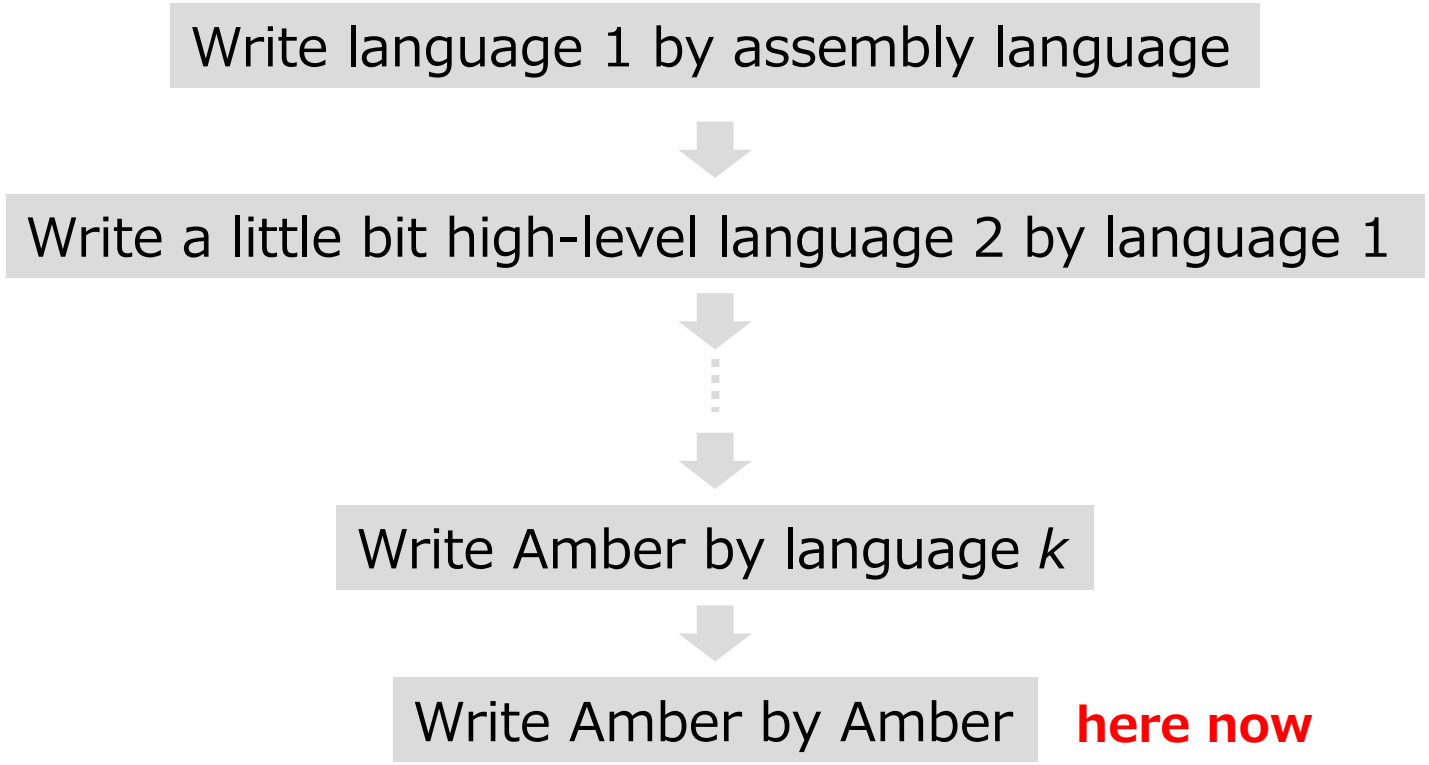
- I wanted to enjoy the creation process itself.
- How could I?

Let’s play with limitations

1. Use assembly language only.
2. No libraries.
3. No code generators.



Strategy : Bootstrapping


Write language 1 by assembly language
Write a little bit high-level language 2 by language 1
Write Amber by language *k*
Write Amber by Amber
here now

What’s the point?

- For fun.
- To cultivate knowledge, techniques, know-hows of compiler-writing.
  - But it’s not cost-effective study method...
- To feel a sense of gratitude and respect for predecessors.

I’ll show the outline of my development process.

1. Created “rowl0” by assembly language

Made a little bit high-level lang. more than asm.

- language name: rowl0
- compiler name: rlc

From regular expressions of tokens

```
/*
 * regular expressions:
 * spaces     : [\n\r\t ]
 * letter     : [A-Za-z_]
 * digit      : [0-9]
 * integer    : 0|[1-9][0-9]*
 * identifier : {letter}({letter)|{digit})*
 * escape     : \\['"?\\abfnrtv0]
 * character  : \'({escape}|[^\\\'\n])\'
 * string     : \"({escape}|[^\\\"\n])*\"
 * comment    : (% [^"%)"]* %)
 * symbol     : .
 */
```

Wrote a state transition diagram

/*
* state transition diagram:
*
* +---+
* |@n | == accepting state
* +---+
*
* +---+   0   +---+
* | 0 +--+-------->|@1 |
* +---+ | +---+
* | [1-9] +---+
*   +------->|@2 +--+
* | +---+ | [0-9]
* | ^ |
* | +----+
* | letter +---+   others   +---+
*   +------->| 3 +--+-------------->|@17| (detect reserved words here)
* | +---+ | +---+
* | ^ | letter|digit
* | +----+
* | \'   +---+ [^\\\'\n] +---+   \'   +---+
*   +------->| 4 +---------->| 5 +-------->|@6 |
* | +-+-+   +---+   +---+
* | | +---+ ['"?\\abfnrtv0] +---+ ^

Converted to jump table

```
/* jump table */
.section .rodata
/*
 *              N   I   S   Z   N   N   S   N   S   D   B   Q   S   N   P
 *              U   N   P   E   O   O   P   :   Q   Q   A   U   Y   L   _
 *              L   V   A   R   N   R   E   :   U   U   C   E   M   :   O
 *              L   A   C   O   Z   M   C   :   O   O   K   S   B   :   R
 *              :   L   E   :   E   A   I   :   T   T   S   T   O   :   _
 *              :   I   S   :   R   L   A   :   E   E   L   I   L   :   X
 *              :   D   :   :   O   C   L   :   :   :   A   O   :   :   :
 *              :   :   :   :   :   H   C   :   :   :   S   N   :   :   :
 *              :   :   :   :   :   :   H   :   :   :   H   :   :   :   :
 */
s0_next:   .long s14,s15,s13, s1, s2, s3, s3, s3, s4, s9,s15,s12,s12,s13,s19
s1_next:   .long s16,s15,s16,s15,s15,s15,s15,s15,s16,s16,s16,s16,s16,s16,s15
s2_next:   .long s16,s15,s16, s2, s2,s15,s15,s15,s16,s16,s16,s16,s16,s16,s15
s3_next:   .long s17,s15,s17, s3, s3, s3, s3, s3,s17,s17,s17,s17,s17,s17, s3
s4_next:   .long s15,s15, s5, s5, s5, s5, s5, s5, s5, s5, s7, s5, s5, s5, s5
s7_next:   .long s15,s15,s15, s8,s15,s15, s8, s8, s8, s8, s8,s15,s15,s15
s9_next:   .long s15,s15, s9, s9, s9, s9, s9, s9, s9,s10,s11, s9, s9, s9, s9
s11_next:  .long s15,s15,s15, s9,s15,s15, s9, s9, s9, s9, s9,s15,s15,s15
s19_next:  .long s17,s15,s17,s20,s20, s3, s3, s3,s17,s17,s17,s17,s17,s17, s3
s20_next:  .long s16,s15,s16,s20,s20, s3, s3, s3,s16,s16,s16,s16,s16,s16, s3
```

And wrote the lexer

```
.global _lex
_lex:
    pushl   %ebp
    movl    %esp, %ebp
    cmpl    $1, unputted
    je      ret_unputted
s0:
    movl    $0, token_len
    movl    $0, token_val
    call    _lex_lookahead
    movl    s0_next(,%eax,4), %eax
    jmp     *%eax
s1:
    call    _lex_consume
    call    _lex_lookahead
    movl    $TOK_INT, token_tag
    movl    s1_next(,%eax,4), %eax
    movl    $0, token_val
    jmp     *%eax
s2:
    call    _lex_consume
    subl    $'0, %eax       /* convert to value */
    movl    token_val, %ebx
    imul    $10, %ebx
```

Wrote rowl0's syntax by BNF

```
Item
    : ToplevelExpr
    | 'return' OrExpr
    | 'if' '(' OrExpr ')' Block
    | 'if' '(' OrExpr ')' Block 'else' Block
    | 'goto' Identifier
    | 'label' Identifier
    | 'export' SimpleItem
    | 'syscall' SimpleItem
    | 'allocate' SimpleItem
    ;

ToplevelExpr
    : Identifier ':' OrExpr
    | Identifier '=>' (Integer|Identifier)
    | Identifier '=' OrExpr
    | Identifier '(' Arguments')'
    | SimpleItem '[' OrExpr ']' '=' OrExpr
    | '*' PrefixExpr '=' OrExpr
    ;

OrExpr
    : XorExpr ('|' XorExpr)*
    ;
```

Then wrote the parser

- recursive descent method

```
.global _item
_item:
    cmpl    $TOK_GOTO, token_tag
    je      1f
    cmpl    $TOK_LABEL, token_tag
    je      2f
    cmpl    $TOK_RETURN, token_tag
    je      3f
    cmpl    $TOK_SYSCALL, token_tag
    je      4f
    cmpl    $TOK_EXPORT, token_tag
    je      5f
    cmpl    $TOK_IF, token_tag
    je      6f
    cmpl    $TOK_ALLOCATE, token_tag
    je      9f
    cmpl    $TOK_WHILE, token_tag
    je      10f
    cmpl    $TOK_WCH, token_tag
    je      11f
```

Generates codes together with parsing

```
_additive_expr:
    call    _multiplicative_expr
3:
    call    _lex
    cmpl    $'+, token_tag
    je      1f
    cmpl    $'-, token_tag
    je      2f
    call    _lex_unput
    ret
1:
    call    _pushl_eax
    call    _lex
    call    _multiplicative_expr
    call    _popl_ebx
    call    _addl
    call    _ebx
    call    _comma
    call    _eax
    call    _nl
    jmp     3b
2:
    call    _pushl_eax
    call    _lex
```

parsing

code generation

- writing memory management is difficult here.
- generates codes without building syntax trees.

Completed the first language “rowl0”!

```
(% p0: input channel %);
fgetc: (p0) {
    allocate(1);
    fill_inbuf(p0);
    if (p0[ICHAN_BEG] == p0[ICHAN_END]) {
        return EOF;
    };
    x0 = rch(p0[ICHAN_BUF], p0[ICHAN_BEG]);
    p0[ICHAN_BEG] = p0[ICHAN_BEG] + 1;
    return x0;
};

getc: () {
    return fgetc(stdin);
};

(% looks ahead next character
 % p0: input channel
 %);
fnextc: (p0) {
    fill_inbuf(p0);
    if (p0[ICHAN_BEG] == p0[ICHAN_END]) {
        return EOF;
    };
```

- no symbol tables.
  - function params must be p0,p1,p2,...
  - to use local variables, allocate stack mems by “allocate(n)” then use x0,x1,x2,...

2.Created a LISP “rowl-core” by “rowl0”

Made a LISP temporarily

- language name: rowl-core
- interpreter name: rlci

- easy to implement
- productivity improvement

Wrote lexer and parser

```
lex: () {
    allocate(2);
    if (unputted) {
        unputted = FALSE;
        return toktag;
    };
label s0;
    if (lookahead() == CH_SPACES) {
        consume();
        goto &s0;
    };
    if (lookahead() == CH_NL) {
        consume();
        goto &s0;
    };
    init_tokbuf();
    goto s0next[lookahead()];
label s1;
    accept(TOK_END);
    return TOK_END;
label s2;
    consume();
    goto s2next[lookahead()];
```

Writing became more comfortable

```
parse_sexp: (p0) {
    allocate(3);
    if (p0 == '`') {
        return mkquote(parse_sexp(lex()));
    };
    if (p0 == '@') {
        return mkunquote(parse_sexp(lex()));
    };
    if (p0 == '(') {
        x0 = parse_list_body(lex());
        eatchar(')',lex());
        return x0;
    };
    if (p0 == TOK_CHAR) {
        return mkchar(token_val());
    };
    if (p0 == TOK_INT) {
        return mkint(token_val());
    };
    if (p0 == TOK_STRING) {
        x0 = unescape(token_text());
        return mkstring(x0);
    };
    if (p0 == TOK_SYMBOL) {
```

Wrote *eval*

```
eval_cons: (p0) {
    allocate(2);
    x0 = car(p0);
    x1 = eval_sexp(x0);
    if (x1 == var_sym)     { return eval_var(cdr(p0)); };
    if (x1 == set_sym)     { return eval_set(cdr(p0)); };
    if (x1 == if_sym)      { return eval_if(cdr(p0)); };
    if (x1 == cond_sym)    { return eval_cond(cdr(p0)); };
    if (x1 == while_sym)   { return eval_while(cdr(p0)); };
    if (x1 == do_sym)      { return eval_do(cdr(p0)); };
    if (x1 == lambda_sym)  { return eval_lambda(cdr(p0)); };
    if (x1 == macro_sym)   { return eval_macro(cdr(p0)); };
    if (x1 == foreach_sym) { return eval_foreach(cdr(p0)); };
    if (x1 == import_sym)  { return eval_import(cdr(p0)); };
    if (prim_p(x1) != nil_sym)   { return (prim_funptr(x1))(eval_args(cdr(p0))); };
    if (lambda_p(x1) != nil_sym) { return eval_applambda(x1, cdr(p0)); };
    if (macro_p(x1) != nil_sym)  { return eval_appmacro(x1, cdr(p0)); };
label eval_cons_error;
    fputs(stderr, "ERROR: invalid application of '");
    pp_sexp(stderr, x1);
    fputs(stderr, "'\n");
    exit(1);
};
```

No memory management

- *mmap* and *munmap* is the only function

~~malloc~~ ~~free~~

1. Does not recovery garbage memories
2. Allocates fresh memories for new objects
3. So, it will die eventually

- When it can compile the next generation compilers, it’s no problem.

Completed a LISP “rowl-core”!

```
; primitive functions
(define assign_indices (table) (do
    (var idx 0)
    (map (lambda (ent) (do
        (incr idx)
        `(@ent @(- idx 1) @(tosym (++ "prim_" ent)))
        )) table)
    ))
```

```
;=== loops ===
(rewrite upto (it beg end body) `(do
    (var @it @beg)
    (var end_ @end)
    (while (<= @it end_) (do
        @body
        (set @it (+ @it 1))
        )
    )))
```

- rich functions
  - *lambda*, *map* etc.
- macros

3.Created a language to write “VM” by “rowl-core”

Decided to create a VM for the next generation

- Created a language just for writing the virtual machine.

- Defined it as a DSL in the LISP “rowl-core”
  - No need of writing lexer and parser!

Wrote the compiler like this

```
(define emit_operand (opd)
    (cond
        ((int? opd)     (print '$' opd))
        ((== opd '\0')  (print "$0"))
        ((char? opd)    (print "$'" (escape opd)))
        ((== `%top opd) (print "(%esp)"))
        ((symbol? opd)  (print opd))
        ((== `var (car opd)) (do
            (var offs (- var_pos (cadr opd)))
            (print (* -4 offs))
            (print '(')
            (emit_operand `%ebp)
            (print ')')
            ))
        ((== `global (car opd)) (do
            (var type (cadr opd))
            (var sym  (caddr opd))
            (if (address? type) (print '$' sym) (print sym))
            ))
        ((== `arg (car opd)) (do
            (print (* 4 (+ (cadr opd) 1)))
            (print '(')
            (emit_operand `%ebp)
            (print ')')
```

Now I could use higher-order functions

```
(define dispatch (callbacks default args) (do
    (if (cons? (car args))
        (do
            (var head (caar args))
            (var fun (assoc head callbacks))
            (if fun
                (fun . args)
                (default . args)
            )
        )
        (default . args)
    )))
```

- productivity was improved a lot

4.Created a virtual machine “rlvm” by the DSL

Wrote codes of VM with the DSL like this

```
(var vm-main-code `(

(fun strlen (str) (
    (int len 0)
    (while (!= (*8 str) '\0') (
        (incr str)
        (incr len)
        ))
    (return len)
    ))

(export memset)
(fun memset (ptr val size) (
    (for i 0 size (
        ([]=8 ptr i val)
```

```
(vtable_push)
(compile vm-main-code)
(vtable_pop)
```

Wrote a garbage collector

```
(fun garbage_collect () (
    (if (|| @IN_GC stop_GC) (
        (alloc_new_block)
        (return)
        ))
    (insert_padding) ; fill residual area in current page
    (= next_gen @(ADVANCE_GENERATION current_gen))
    (= page_queue_head 0)
    (= page_queue_tail 0)
    (= num_allocated_pages 0)
    (= num_seq 0)

    ; promote all pages which are refered from machine stack and vm-stack
    ; NB: scanning of machin registers is not necessary because
    ;     %ebx (program counter) will be stored in vm-stack and
    ;     other registers are not callee-save registers.
    (scan_machine_stack)
    (scan_vm_stack)
    (scan_exception_stack)
    (scan_reset_stack)
    (scan_continuation_list)

    (move_exception_stack)
```

- Copying GC
- Cheney’s algorithm

Wrote primitive functions

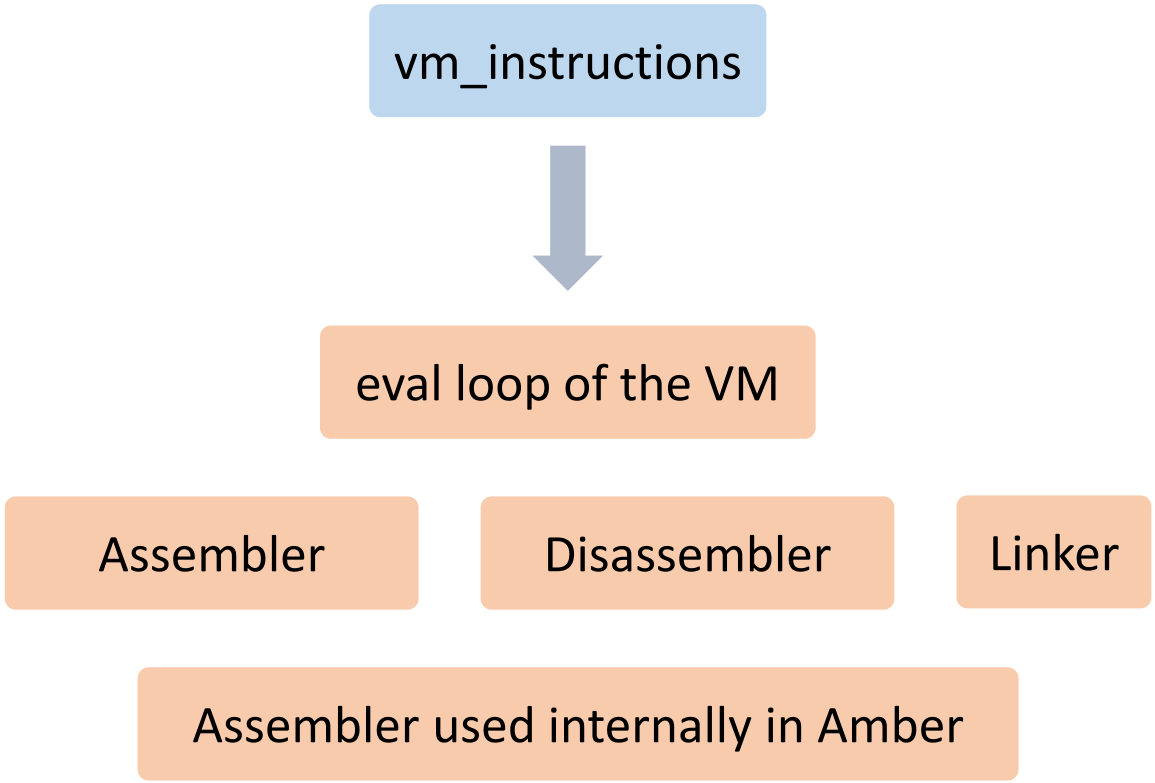
```
(export prim_output_string)
(fun prim_output_string (ochan str) (
    (if (== (get ochan 1) @STDERR_FD) (
        ; Do not buffer outputs to stderr
        (syscall @SYS_WRITE @STDERR_FD str (strlen str))
        (return)
        ))
    (while (!= (*8 str) '\0') (
        (prim_output_char ochan (*8 str))
        (incr str)
        ))
    ))

(char[] 10 output_int_digits)
(export prim_output_int)
(fun prim_output_int (ochan num) (
    (if (< num 0) (
        (if (== num 0x80000000) (
            ; neg num == neg
            (prim_output_string ochan "-2147483648")
            (return)
            ))
        (prim_output_char ochan '-')
        (prim_output_int ochan (neg num))
```

An application of meta-programming

- The table of instructions of the VM

```
(var vm_instructions `(
    (nop   () @true ())
    (drop  () @true ((asm "addl $4, %esi")))
    (drop2 () @true ((asm "addl $8, %esi")))
    (dup   () @true ((asm "movl (%esi), %eax") (vmpush %eax)))
    (swap  () @true (
        (asm "movl (%esi), %eax")
        (asm "movl 4(%esi), %ecx")
        (asm "movl %eax, 4(%esi)")
        (asm "movl %ecx, (%esi)")
        ))
    ; 'rot:1' 'number of elements:1' 'shift:1'
    (rot (byte byte) @true (
        (vmubyte 1 %edx)
        (vmubyte 2 %ecx)
        (asm "shll $2, %edx")
        (asm "movl %esi, %eax")
        (asm "addl %edx, %eax")
```

Generates various codes from the table



- reflects changes of instructions automatically
- It is very easy to make this kind of mechanism with LISP

Wrote instruction sets

```
(nop   () @true ())
(drop  () @true ((asm "addl $4, %esi")))
(drop2 () @true ((asm "addl $8, %esi")))
(dup   () @true ((asm "movl (%esi), %eax") (vmpush %eax)))
(swap  () @true (
    (asm "movl (%esi), %eax")
    (asm "movl 4(%esi), %ecx")
    (asm "movl %eax, 4(%esi)")
    (asm "movl %ecx, (%esi)")
    ))
; 'rot:1' 'number of elements:1' 'shift:1'
(rot (byte byte) @true (
    (vmubyte 1 %edx)
    (vmubyte 2 %ecx)
    (asm "shll $2, %edx")
    (asm "movl %esi, %eax")
    (asm "addl %edx, %eax")

    (asm "1:")
    (asm "testl %ecx, %ecx")
    (asm "jz 2f")
    (asm "subl $4, %eax")
    (asm "pushl (%eax)")
    (asm "subl $1, %ecx")
```

Floating point arithmetics

```
(fadd () @true (
    (vmpop %ecx)
    (asm "movl (%esi), %eax")
    (asm "fldl (%ecx)")
    (asm "fldl (%eax)")
    (asm "faddp %st, %st(1)")
    (asm "fstpl (%eax)")
    ))
(fsub () @true (
    (vmpop %ecx)
    (asm "movl (%esi), %eax")
    (asm "fldl (%ecx)")
    (asm "fldl (%eax)")
    (asm "fsubp %st, %st(1)")
    (asm "fstpl (%eax)")
    ))
(fmul () @true (
    (vmpop %ecx)
    (asm "movl (%esi), %eax")
    (asm "fldl (%ecx)")
    (asm "fldl (%eax)")
    (asm "fmulp %st, %st(1)")
    (asm "fstpl (%eax)")
    ))
```

Multi-precision integer arithmetics

```
; bigint arithmetic
; 1st argument must be big-integer
; 2nd argument must be big-integer or fixed-integer
; no run-time check is performed
(badd   () @true (
    (asm "pushl %ebx")

    (asm "movl (%esi), %eax") ; rhs
    (asm "movl 4(%esi), %ebx") ; lhs
    (asm "movl 12(%eax), %eax") ; sign of rhs
    (asm "cmpl 12(%ebx), %eax")
    (asm "jne bsub_main") ; if rhs and lhs has different signs, switch to bsub
    ; compute (stp) += (eax)
    (asm "badd_main:")
    (asm "movl (%esi), %eax")
    (asm "movl 4(%esi), %ebx")
    (asm "movl 4(%ebx), %ecx")  ; number of digits of lhs
    (asm "cmpl 4(%eax), %ecx")
    (asm "cmovl 4(%eax), %ecx")
    (asm "addl $1, %ecx") ; ecx = max(lhs.ndigit, rhs.ndigit) + 1
    (asm "cmpl 8(%ebx), %ecx") ; compare capa of lhs and ecx
    (asm "jle 2f")
    ; reserve array of lhs
    (asm "pushl %eax")
```

Exception handling

```
; pop unwind stack
(unwind_pop () @true (
    (asm "movl exstack, %eax")
    (asm "movl 12(%eax), %eax")
    (asm "movl %eax, exstack")
    ))
; throw given value
(throw () @nil (
    (vmpop %ecx)    ; exception value
    (asm "movl exstack, %eax")
    (asm "movl (%eax), %ebx")
    (asm "movl 4(%eax), %esi")
    (asm "movl 8(%eax), %edi")
    (asm "movl 12(%eax), %eax")
    (asm "movl %eax, exstack")
    (vmpush %ecx)
    (vmfetch)
    ))
```

Delimited Continuation

```
(shift () @nil (
    (vmsave)
    (asm "movl resetstack, %edx")
    (asm "pushl (%edx)")    ; stack top at `reset'
    (asm "movl %esi, %eax")
    (asm "addl $4, %eax")
    (asm "pushl %eax")      ; stack top at `shift'
    (asm "movl %ebx, %eax")
    (asm "addl $1, %eax")
    (asm "pushl %edi")      ; base pointer
    (asm "pushl %eax")      ; program counter
    (call prim_make_continuation)
    (asm "addl $16, %esp")
    (vmrestore)
    (vmpop %ebx)
    (asm "addl $12, %ebx") ; pointer to the bytecode

    (asm "movl resetstack, %edx")
    (asm "movl 8(%edx), %ecx")
    (asm "movl 4(%edx), %edi")
    (asm "movl (%edx), %esi")
    (asm "movl 12(%edx), %edx")
    (asm "movl %edx, resetstack")
```

Completed the virtual machine “rlvm”!

- 186 instructions
- stack machine
- copying GC
- exception handling
- shift/reset delimited continuation
- floating-point arithmetics, multi-precision arithmetics

5. Created a tool chain for “rlvm”

There was no programming tools for “rlvm”

- Created a tool chain for the VM
  - a programming language “rowl1”
  - its compiler
  - assembler
  - disassembler
  - linker

Wrote “rowl1”, assembler and compiler

- Defined as a DSL of “rowl-core”

```
(define compile_expr (expr drop) (do
    (cond
        ((== `nop expr)    (if (not drop) (emit_insn `imm_i0)))
        ((== `null expr)   (if (not drop) (emit_insn `imm_i0)))
        ((== `return expr) (do (assert "return" drop) (emit_insn `return)))
        ((== `break expr) (do (assert "break" drop) (compile_break expr)))
        ((== `continue expr) (do (assert "continue" drop) (compile_continue expr)))
        ((== `getself expr) (if (not drop) (emit_insn `self)))
        ((char? expr) (compile_int_lit (char2int expr) drop))
        ((int? expr) (compile_int_lit expr drop))
        ((string? expr) (compile_string_lit expr drop))
        ((symbol? expr) (compile_var expr drop))
        ((== `var (car expr)) (compile_local_vardecl expr drop))
        ((== `address (car expr)) (compile_addressof expr drop))
        ((== `if (car expr)) (compile_if expr drop))
        ((== `while (car expr)) (compile_while expr drop))
        ((== `for (car expr)) (compile_for expr drop))
        ((== `rfor (car expr)) (compile_rfor expr drop))
        ((== `lfor (car expr)) (compile_list_for expr drop))
        ((== `tswitch (car expr)) (compile_tswitch expr drop))
        ((== `lswitch (car expr)) (compile_lswitch expr drop))
        ((and (== `try (car expr)) (== `catch (caaddr expr))) (compile_try expr drop))
        ((== `throw (car expr)) (compile_throw expr drop))
        ((== `label (car expr)) (compile_label expr drop))
```

Wrote linker and disassembler

- Wrote these tools by “rowl1”, so they run on “rlvm”
- The linker requires GC since it uses a lot of memory

```
(fun output_executable (file) (
    (var oc (open_out file @(| O_WRONLY O_CREAT O_TRUNC) @TRUE))
    (if (! oc) (error "open failed"))

    (output_word oc @EXE_MAGIC)

    (output_word oc total_nref)

    (output_word oc total_vsize)
    (for i 0 (vector_size values) (do
        (output_bytes oc (vector_at values i) (ivector_at value_sizes i))
        ))

    (output_word oc total_len)
    (output_startup oc)
    (for i 0 (vector_size codes) (do
        (var codelen (ivector_at code_lens i))
        (output_bytes oc (vector_at codes i) codelen)
        ))

    (close_out oc)
```

Example outputs of the disassembler

```
==== disasm result of "linker" ====
- number of global objects = 8
- static values = 274 bytes
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : ................
00000010: 00 00 00 00 73 74 61 72 74 75 70 2e 72 6c 6f 00 : ....startup.rlo.
00000020: 2d 6f 00 61 2e 72 6c 78 00 69 6e 76 61 6c 69 64 : -o.a.rlx.invalid
00000030: 20 66 69 6c 65 20 66 6f 72 6d 61 74 31 00 69 6e :  file format1.in
00000040: 76 61 6c 69 64 20 66 69 6c 65 20 66 6f 72 6d 61 : valid file forma
00000050: 74 32 00 69 6e 76 61 6c 69 64 20 66 69 6c 65 20 : t2.invalid file
00000060: 66 6f 72 6d 61 74 33 00 6f 70 65 6e 20 66 61 69 : format3.open fai
00000070: 6c 65 64 00 69 6e 76 61 6c 69 64 20 66 69 6c 65 : led.invalid file
00000080: 20 66 6f 72 6d 61 74 20 28 6d 61 67 69 63 20 6e :  format (magic n
00000090: 75 6d 62 65 72 20 6d 69 73 6d 61 74 63 68 29 00 : umber mismatch).
000000a0: 6c 6f 61 64 20 66 61 69 6c 65 64 00 6c 6f 61 64 : load failed.load
000000b0: 20 66 61 69 6c 65 64 00 75 6e 64 65 66 69 6e 65 :  failed.undefine
000000c0: 64 20 73 79 6d 62 6f 6c 00 75 6e 64 65 66 69 6e : d symbol.undefin
000000d0: 65 64 20 73 79 6d 62 6f 6c 00 75 6e 64 65 66 69 : ed symbol.undefi
000000e0: 6e 65 64 20 73 79 6d 62 6f 6c 00 6d 61 69 6e 00 : ned symbol.main.
000000f0: 75 6e 64 65 66 69 6e 65 64 20 73 79 6d 62 6f 6c : undefined symbol
00000100: 20 6d 61 69 6e 00 6f 70 65 6e 20 66 61 69 6c 65 :  main.open faile
00000110: 64 00
- instructions = 1919 bytes
0       : imm_i0
1       : pcall make_vector 1
5       : storeo 0
8       : imm_i0
9       : pcall make_ivector 1
13      : storeo 1
```

Ready to program on “rlvm”!

- writing programs for rlvm
- disassembling of byte-codes
- supports separate compilation

- Reached the starting line

6.Wrote “Amber” by “rowl1”

Started developing “Amber”

- dynamic scripting language
- instance-based object-oriented system
- run on rlvm

Wrote an assembler

```
(export fun make_assembler () (
    (return (variant @AssemblerE 5 (make_cvector 0) (make_ivector 0) 0 (ma
    ))

(export fun get_code (c) (
    (if (field_get c 3) (replace_labels c))
    (var code (field_get c 1))
    (var addrs (field_get c 4))
    (var objs (field_get c 5))
    (return (make_bytecode (cvector_raw code) (cvector_size code)
        (ivector_raw addrs) (vector_raw objs) (ivector_size addrs)))
    ))

(export fun get_address (c) (
    (return (cvector_size (field_get c 1)))
    ))

; aliases
(export fun put_zero (c)   ((put_imm_i0 c)))
(export fun put_undef (c)  ((put_imm_int c @C_UNDEF)))
(export fun put_nil (c)    ((put_imm_int c @C_NIL)))
(export fun put_true (c)   ((put_imm_int c @C_TRUE)))
(export fun put_false (c)  ((put_imm_int c @C_FALSE)))
(export fun put_car (c)    ((put_field_get0 c)))
(export fun put_cdr (c)    ((put_field_get1 c)))
(export fun put_setcar (c) ((put_field_set0 c)))
(export fun put_setcdr (c) ((put_field_set1 c)))
```

- The former assembler assembles codes ahead of time and run on rlci
- This assembler assembles codes just in time and run on rlvm
  - fills addresses by backpatching

Wrote the object system

- slots, messages and parent delegation

```
(fun set_slot_ (obj sym val overwrite union alias) (
    (if (== sym Parent) (return (set_parent obj val)))
    (var ref (get_slot_ref obj sym @C_TRUE))
    (if (&& (!= overwrite @C_FALSE) (== ref @C_UNDEF))
        (if (== (node_head obj) VarTable)
            (throw (undefined_var sym))
            (throw (undefined_slot obj sym))
            )
        )
    (var tbl (get_slot_table sym))
    (if (== ref @C_UNDEF) (do
        (wrtable_add tbl obj val)
        (return val)
        ))
    (var v (field_get ref 3))
    (if (&& (!= union @C_FALSE) (&& (is_function v) (is_function val))) (do
        (var f (union_function val v))
        (if (!= overwrite @C_FALSE)
            (field_set ref 3 f)
            (wrtable_add tbl obj f)
            )
        (return f)
        ))
```

Wrote Amber’s core feature on the system

- dynamic pattern-matching engine

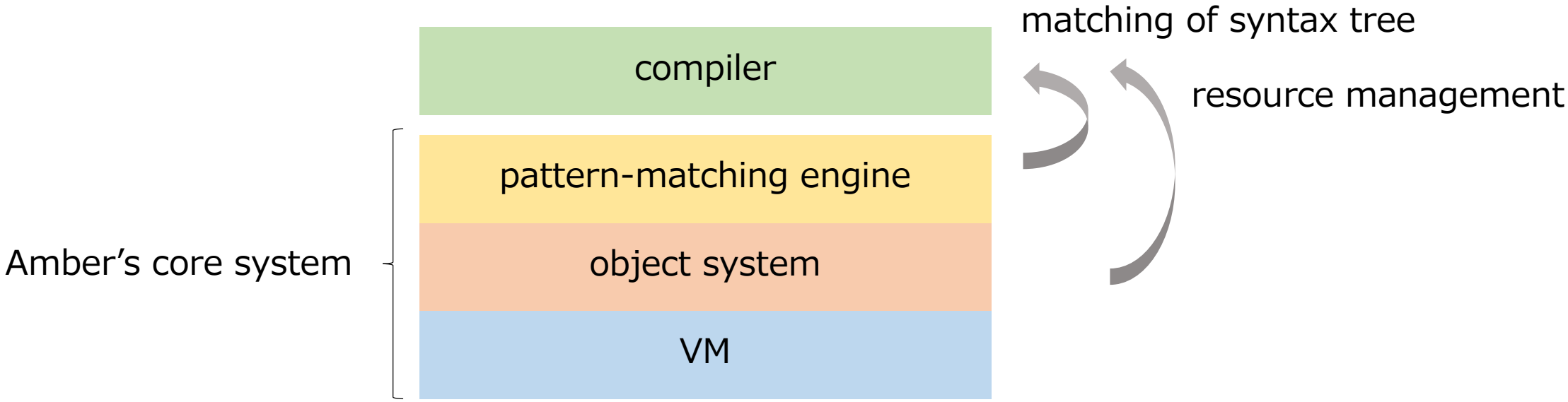
```
(export fun compile_matching (obj def) (
    (if (! (is_function def))
        (throw (type_error (string "function") def))
        )

    (var cache (field_get def 1))
    (if cache (return cache))

    (var asm (make_assembler))
    (var exit_lbl (fresh_label asm))

    (var nocheck (compile_matching_function asm obj def exit_lbl))
```

- mechanism of partial function fusion

```
(export fun union_function (fun1 fun2) (
    (if (! (is_function fun1)) (throw (type_error (string "function") fun1)))
    (if (! (is_function fun2)) (throw (type_error (string "function") fun2)))
    (return (variant @BoundFunctionE 3 0 fun1 fun2))
    ))
```

Wrote the compiler

- Made Amber compiler as one of Amber objects

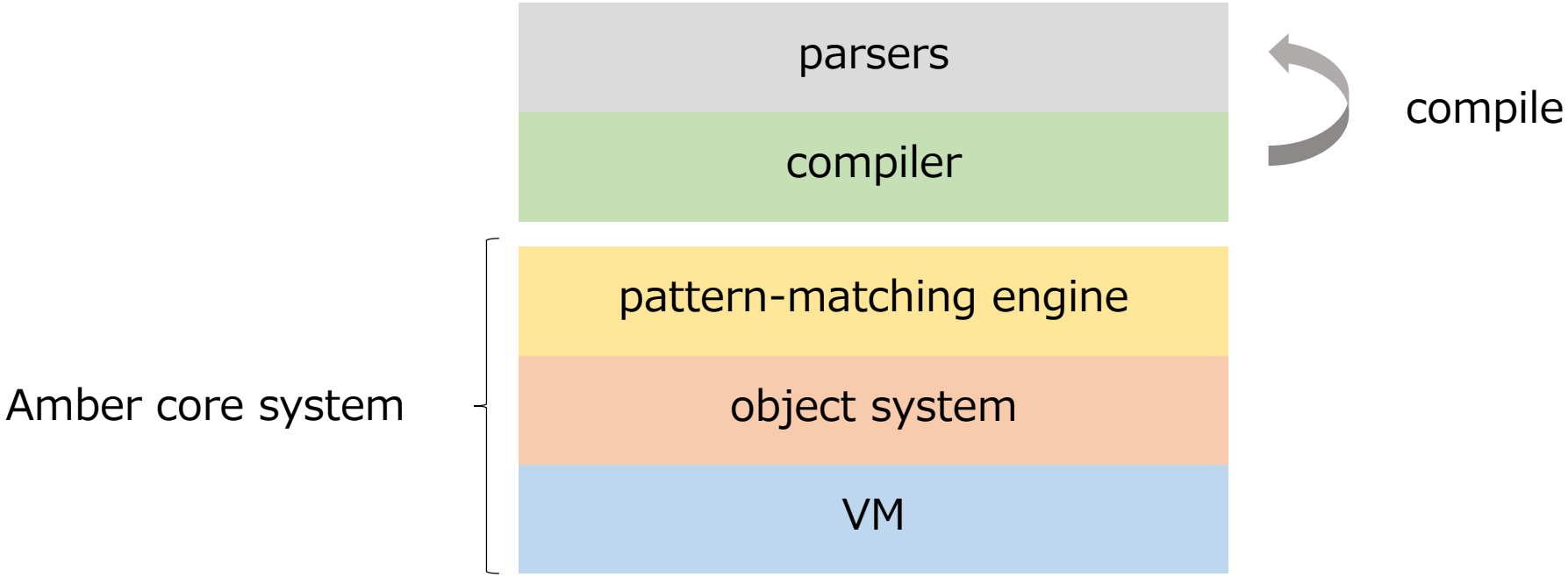


Wrote closure-conversion

```
(fun compile_function (asm_body asm_guard args body guard cls) (
    ;; closure conversion
    (var tbl (make_symtable 10))
    (symtable_push tbl)
    (var vars (collect_params args))
    (while vars (do
        (symtable_add tbl (car vars) maybe)
        (= vars (cdr vars))
        ))
    (if (!= cls @C_NIL)
        (symtable_add tbl cls maybe)
        )
    (= body (scan_clsvars body tbl))

    ; this function needs to allocate an environment for closures.
    (var newcls (to_unique_sym "closure")) ; closures' environment.
    (var clssize (setup_closure_env tbl newcls))
    (if (> clssize 0) (do
        (= body (subst_clsvars tbl newcls body))

        ; allocate closures' environment.
        (var stmts 0)
        (= stmts (cons (defvar newcls
            (make_node1 AllocClsEnv (box clssize))) stmts))
        (if (== cls @C_NIL)
            (= stmts (store_args tbl args stmts))
            (= stmts (store_args tbl (cons cls args) stmts))
            )
```

Wrote parsers

- compiles parsers at run-time
- each parser is a usual Amber object (closure)



very simple syntax

1. literals are *expressions*
2. for a symbol *h* and expressions *e1*,..,*en* (n>=0), *h{e1, ..., en} is an expression*
3. no other form of Amber's expression

```
nineties@euler% amber
amber:1> Apply{puts, List{"hello world"}}
hello world
=> nil
amber:2>
```

Used Packrat parsing method

```
(fun parse_fullform_ (p) (
    (var c (lookahead p))
    (if (< c 0) (return (mayparse p 0)))
    (if (! (is_letter c)) (return (noparse p 0)))

    (var state (copy_parser_state p))

    (if (== (parse_symbol p) @C_FALSE) (return @C_FALSE))

    (var hd @(TOKEN p))
    (parse_char p '{')
    @(CHECK p)
    (parse_spaces p)

    (var args 0)
    (if (!= (lookahead p) '}') (do
        (while (> (lookahead p) 0) (do
            (parse "expr" p)
            @(CHECK p)
            (= args (cons @(TOKEN p) args))
            (parse_spaces p)
            (if (== (lookahead p) '}') break)
            (parse_char p ',')
            @(CHECK p)
            (parse_spaces p)
            ))
        ))
    (parse_char p '}')
```

- scanner less

Encoding/decoding floating-point literal was difficult

- wrote them by my self
  because of “no libc” limitation

- require multi-precision integer arithmetic which I wrote before

Amber interpreter is completed!

- dynamic scripting language
- run on rlvm

- instance-based object oriented system
- dynamic pattern-matching engine
- partial function fusion
- lexical closure

- I got modern programming language!

7. Created Amber’s standard library

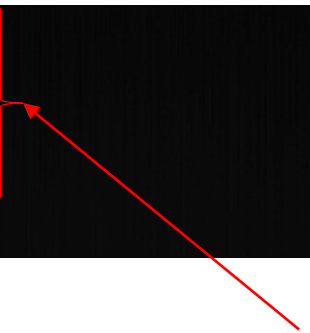
Amber has strong self extensibility

- Amber’s simple syntax is extended in a standard library
  - amber/lib/syntax/parse.ab

- Builds its syntax during boot sequence

Only has very simple syntax at first

```
" Copyright (C) 2010 nineties "
" $Id: syntax/parse.ab 2014-05-11 20:46:49 nineties $ "

" Syntax definition of the Amber language. "

" Enable comments. "
Assign{Var{syntax, comment}, Slot{prim, parse_shell_style_comment}}
```



used string literal for comments
because there is no syntax for comments

Defines a syntax for defining syntaxes

```
DefineSyntax{
    character_set,
    List{Tuple{List{Many1{Choice{List{"\\", any}, List{Not{"]"}, any}}}},
        Apply{Slot{prim, input_text}, List{parser, begin, end}}}
    }
}

DefineSyntax{
    primary_syntax_element,
    List{
        Tuple{List{"."}, Quote{any}},
        Tuple{List{string}, node0},
        Tuple{List{identifier}, node0},
        Tuple{List{"(", syntax_element, ")"}, node1},
        Tuple{List{"(", Many1{syntax_element}, ")"}, node1, multiline},
        Tuple{List{"[", "^", character_set, "]"},
            Quasiquote{NoneOf{Unquote{node2}}}},
        Tuple{List{"[", character_set, "]"},
            Quasiquote{OneOf{Unquote{node1}}}}
    }
}

DefineSyntax{
    syntax_element,
    List{
        Tuple{List{primary_syntax_element, "?"},
            Quasiquote{Option{Unquote{node0}}}},
        Tuple{List{primary_syntax_element, "*"},
```

Defines Amber’s syntax with the syntax

```
### Expressions
block_body ::= aligned(stmt) { Node{Quote{Block}, node0} }

primary_block ::= <<multiline>> "{" block_body "}" { node1 }

block ::= primary_block
        | block_body

primary_expr
    ::= primary_block
      | <<multiline>> "<" postfix_expr "," postfix_expr ">" "{"
         parsewith(Var{Apply{eval, List{node1}}, syntax}, node3) "}"
        { node6 }
      | <<multiline>> "<" postfix_expr ">" "{"
         parsewith(Var{Apply{eval, List{node1}}, syntax}, Quote{default}) "}"
        { node4 }
      | <<multiline>> "(" stmt ")"            { node1 }
      # List literal
      | <<multiline>> "[" delimited(expr, ",") "]"  { node1 }
      # Tuple literal
      | <<multiline>> "(" delimited2(expr, ",") ")"
        { Node{Quote{Tuple}, node1} }
      | "|" expr "|"
        { Node{Quote{Abs}, List{node1}} }
      | 'self           { Quote{Self{}} }
      | "." symbol
        { Node{Quote{Slot}, List{Quote{Self{}}, node1}} }
      | fullform
```

Builds macro system

```
### Macro
macro := e -> e
Define{expand_macro_sub, (obj, lvl) -> {
        head := prim.head(obj)
        args := prim.arguments(obj)
        Node{
            prim.head(obj),
            prim.list_map(e -> expand_macro(e, lvl), args)
        }
    }, union}
Define{expand_macro_sub, (obj, _) when prim.atom?(obj) -> obj, union}
Define{expand_macro_sub,
    (obj, lvl) when prim.int_less_than?(lvl, 0) ->
    { throw `LogicError{"Unquotetation outside quasi-quotation"} },
    union}
Define{expand_macro_sub,
    (obj, lvl) when prim.identical?(prim.head(obj), 'Unquote) -> {
        prim.set_arg(obj, 0,
            expand_macro(prim.get_arg(obj, 0), prim.int_sub(lvl, 1)))
        obj
    },
    union}
Define{expand_macro_sub,
    (obj, lvl) when prim.identical?(prim.head(obj), 'Quasiquote) -> {
        prim.set_arg(obj, 0,
            expand_macro(prim.get_arg(obj, 0), prim.int_add(lvl, 1)))
        obj
    },
```

Gives meanings to syntaxes by macros

```
macro(|x|)     = `abs(!x)
macro(+x)      = `uplus(!x)
macro(-x)      = `uminus(!x)
macro(x + y)   = `add(!x, !y)
macro(x - y)   = `sub(!x, !y)
macro(x ++ y)  = `ncadd(!x, !y)
macro(x * y)   = `mul(!x, !y)
macro(x ** y)  = `ncmul(!x, !y)
macro(x ^ y)   = `pow(!x, !y)
macro(x / y)   = `div(!x, !y)
macro(x // y)  = `quo(!x, !y)
macro(x % y)   = `rem(!x, !y)
macro(x | y)   = `union(!x, !y)
macro(x < y)   = `less_than?(!x, !y)
macro(x > y)   = `greater_than?(!x, !y)
macro(x <= y)  = `less_equal?(!x, !y)
macro(x >= y)  = `greater_equal?(!x, !y)
macro(x == y)  = `equal?(!x, !y)
macro(x != y)  = `(not equal?(!x, !y))
macro(x <=> y) = `compare(!x, !y)
macro(x += y)  = `ArithAssign{!x, !y, add!}
macro(x ++= y) = `ArithAssign{!x, !y, ncadd!}
macro(x -= y)  = `ArithAssign{!x, !y, sub!}
macro(x *= y)  = `ArithAssign{!x, !y, mul!}
macro(x **= y) = `ArithAssign{!x, !y, ncmul!}
macro(x ^= y)  = `ArithAssign{!x, !y, pow!}
macro(x /= y)  = `ArithAssign{!x, !y, div!}
macro(x //= y) = `ArithAssign{!x, !y, quo!}
```

Now Amber got rich syntax

```
make_counter() := {
    n := 0
    return () -> { n += 1 }
}

counter1 := make_counter()
counter2 := make_counter()

puts(counter1())
puts(counter1())
puts(counter2())
puts(counter1())
puts(counter2())
```

Extends object system

```
object_system := enter_module('object_system, true)

prim.array_push(syntax::reserved_words, "class")
prim.array_push(syntax::reserved_words, "trait")
postfix_expr ::= 'class symbol "{" delimited(pattern, ",") "}"
                 { `DefClass{!$1, !$3} }
               | 'class symbol
                 { `DefClass{!$1} }
               | 'trait symbol
                 { `DefTrait{!$1} }

macro(DefClass{head, fields}) = `seq {
        Define{!head, 'Class{!head} with .parent = Trait}

        Define{Slot{!head, make}, Lambda{!fields,
            Node{'!head, !fields} with .parent = !head}}

        Define{Slot{!head, make}, Lambda{!prim.list_map(f -> `Keyword{!f, nil}, fields),
            Node{'!head, !fields} with .parent = !head}, union}

        Define{Slot{!head, new}, Slot{!head, make}}

        WithSlots{!head, !prim.list_map_with_index((f, i) ->
            `(Slot{Self{}, !f} = alias(!i)), fields)}
    }

macro(DefClass{head}) = `Define{!head, 'Class{!head} with .parent = Trait}
```

Now Amber got rich object system

- Inheritence, mix-in etc.

```
amber:1> class A{} with {
amber:1~         .name = "A"
amber:1~         .hello() = "This is " ++ .name
amber:1~ }
=> Class{A}
amber:2> class B{} with {
amber:2~         .name = "B"
amber:2~         .parent = A
amber:2~ }
=> Class{B}
amber:3> obj := B.new()
=> B{}
amber:4> obj.hello()
=> "This is B"
```

Now the development is under suspension

- No plans of further updates

- Try following commands to invoke Amber shell (Linux only)
- See the outputs of the make command

```
% git clone https://github.com/nineties/amber.git
% cd amber
% make; sudo make install
% amber
```

Summary

- I could reach relatively high-level language. Feel satisfied.